# Programming Macro Tree Transducers

Patrick Bahr *
Department of Computer Science
University of Copenhagen
paba@diku.dk

Laurence E. Day
Functional Programming Laboratory
University of Nottingham
led@cs.nott.ac.uk

## Abstract

A tree transducer is a set of mutually recursive functions transforming an input tree into an output tree. *Macro* tree transducers extend this recursion scheme by allowing each function to be defined in terms of an arbitrary number of accumulation parameters. In this paper, we show how macro tree transducers can be concisely represented in Haskell, and demonstrate the benefits of utilising such an approach with a number of examples. In particular, tree transducers afford a modular programming style as they can be easily composed and manipulated. Our Haskell representation generalises the original definition of (macro) tree transducers, abolishing a restriction on finite state spaces. However, as we demonstrate, this generalisation does not affect compositionality.

***Categories and Subject Descriptors*** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

***Keywords*** recursion schemes, deforestation, modularity

## 1. Introduction

Tree automata [5] describe restricted forms of computations on trees. The simplest tree automata are known as acceptors, which – analogously to finite state string automata – are used to describe a set of (potentially infinite) trees. Other types of automata, called transducers, describe binary relations between trees, in general, or total functions on trees, in the case of total, deterministic transducers, which we consider in this paper.

Tree automata are widely applied in the fields of logic [31], term rewriting [32], XML processing [16], natural language processing [20]. The restrictions upon such automata affords them many desirable meta-properties such as decidability of equality [5] and reversability [24].

In this paper, we aim to utilise the properties of tree automata in order to structure functional programs. In this sense, this paper follows a line of work [1–3] in which we explore the use of recursion schemes derived from tree automata to build highly modular programs that manipulate tree structures. In particular, our goal is to extend the expressive power of this approach by using *macro tree transducers* (MTTs) [9].

### 1.1 Contributions

In previous work [1], we considered both top-down tree transducers (DTTs) and bottom-up tree transducers (UTTs) [27, 30]. Intuitively, DTTs represent a set of mutually recursive functions, and MTTs generalise this recursion scheme by allowing each function to include an arbitrary number of additional accumulator parameters. In this paper we show how to represent MTTs in Haskell using a free monad type.

In particular, the contributions are as follows:

- Starting from the representation of DTTs in Haskell, we illustrate a restriction upon the state space derived from the types (Section 3.1). In contrast to the original intent of MTTs, this observation is independent from the availability of accumulation parameters.
- We show how this restriction of DTTs in Haskell is mitigated by moving to a polymorphic state space (Section 3.2).
- We illustrate that the generalisation to a polymorphic state space corresponds to the generalisation from DTTs to MTTs in automata theory (Section 4).
- Our representation of tree transducers in Haskell generalises their automata theoretic definitions, since we do not require a finite state space. We illustrate the benefit of this generalisation (Section 2.5) and show that the compositional properties of MTTs and DTTs are maintained in this setting (Section 6).
- We also represent a recursion scheme that combines top-down and bottom-up state propagation, namely *MTTs with regular look-ahead* (Section 7).

The recursion schemes presented in this paper have been implemented as part of the `compdata` Haskell library [4]. This implementation combines the recursion schemes with Swierstra's *data types à la carte* [29] and other modular constructs [1]. The only crucial Haskell extension that we need for our representation is rank-2 polymorphism.

### 1.2 Why Transducers?

Before diving into the technical details, we highlight the benefits of using tree transducers in Haskell in the first place.

As alluded to in the introduction, tree transducers lend themselves to a highly modular programming style. In previous work [1] we have demonstrated that tree transducers allow us to leverage modularity along multiple, independent dimensions.

The primary notion of modularity is that of *sequential composition*. Given two transformations, one of type $A \to B$ and one of type $B \to C$, we are able to construct a single transformation of type $A \to C$ by combining the underlying transducers. This allows

* This work was done while the author was visiting the Functional Programming Laboratory at the University of Nottingham; the work was supported by the Danish Council for Independent Research.

us to split complex transformations – as seen in compilers, for example – into sequential phases and combine them without incurring undue performance penalties. This proves a powerful mechanism when performing deforestation [22, 23, 37], for example.

Secondly, we can modularise the types of the input trees. Given a transducer defined over trees of signature $\mathcal{F}$ and one over $\mathcal{G}$, we can compose the two to obtain a transducer defined over trees of signature $\mathcal{F} \uplus \mathcal{G}$. This is done via the *data types à la carte* [29] technique of Swierstra. But one should note that this construction is not always straightforward, as subtle interactions between signatures may exist. However, our proposed technique allows for a flexible composition that accounts for these problems (see [1] for a number of examples).

The third aspect of modularity is the fact that we can compose simpler automata to obtain more complex ones. For example, a transducer may perform a transformation (e.g. inlining) that is dependent on information that is provided by other automata independently (e.g. information about the occurrences of bound variables). This flexible approach not only reduces the complexity of individual automata, but ensures that they can be substituted easily regardless of the context in which they operate.

Modularity aside, the structure of tree transducers permits other operations that manipulate computations on trees. A simple but powerful class of such operations is the automatic generation and propagation of annotations. For example, given an code generator (as an MTT) as part of a compiler, a lifting can be defined that automatically propagates annotations of the input (e.g. annotations indicating the originating line in a source file) [2]. Section 5 demonstrates this lifting for the case of MTTs.

## 2. Top-Down Tree Transducers

In this section, we briefly explain the concept of top-down tree transducers, which is the type of automata that we shall built upon subsequently.

### 2.1 First-Order Terms

The tree automata that we consider in this paper operate on terms over some signature $\mathcal{F}$, which is a set of function symbols with a fixed arity. We write $f/n \in \mathcal{F}$ to indicate that $f$ is a function symbol in $\mathcal{F}$ of arity $n$. Given a signature $\mathcal{F}$ and some set $\mathcal{X}$, the set of terms over $\mathcal{F}$ and $\mathcal{X}$, denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is the smallest set $T$ such that $\mathcal{X} \subseteq T$ and if $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in T$ then $f(t_1, \ldots, t_n) \in T$. Instead of $\mathcal{T}(\mathcal{F}, \emptyset)$ we write $\mathcal{T}(\mathcal{F})$ and call its elements terms over $\mathcal{F}$. Tree automata run on terms in $\mathcal{T}(\mathcal{F})$.

We will use the words "tree" and "term" interchangeably. In general, we will use the words "term" and "subterm", but in certain contexts it is beneficial to think of the tree representation of a term. For example, we shall use depictions of terms in the form of trees in order to illustrate the workings of the tree automata we consider. In this context it is convenient to talk about *nodes* and *edges* of the tree representation. For example, for a term of the form $f(t_1, \ldots, t_k)$, we think of a tree rooted in a node $n$ labelled by $f$, with $k$ outgoing ordered edges to nodes $n_1, \ldots, n_k$. These nodes are called the *successor nodes* of $n$, and each $n_i$ is the root node of a tree that represents the term $t_i$.

Each of the tree automata that we describe in this paper consists at least of a finite set $Q$ of states and a set of rules according to which input terms are transformed into an output terms. While performing such a transformation, these automata maintain state information, which is stored as annotations in the intermediate results of the transformation. To this end each state $q \in Q$ is considered as a unary function symbol and a subterm $t$ is annotated with state $q$ by writing $q(t)$. For example, $f(q_0(a), q_1(b))$ represents the term $f(a, b)$, where the two subterms $a$ and $b$ are annotated with states $q_0$ and $q_1$, respectively.

The rules of the tree automata in this paper will all be of the form $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F}', \mathcal{X})$, where $\mathcal{F}' = \mathcal{F} \uplus \{q/k_q \mid q \in Q\}$. For the simple tree automata, the arity $k_q$ of each state $q$ will be just 1. We shall see later in Section 4, that this restriction is lifted for macro tree transducers. The rules can be read as term rewrite rules, i.e. the variables in $l$ and $r$ are placeholders that are instantiated with terms when the rule is applied. Running an automaton is then simply a matter of applying these term rewrite rules to a term. The different kinds of tree automata only differ in the set of rules they allow.

In general, tree transducers induce a relation on trees that pairs each input tree with each of its corresponding outputs. That is, tree transducers may be non-deterministic and partial. Since we are only interested in tree transformations that are functions, in particular transformations that occur in compilers, we shall only consider total, deterministic automata. Roughly speaking, "deterministic" means that, in each "situation", there is at most one rule applicable, while "total" means that there is at least one rule applicable.

### 2.2 Top-Down Tree Transducers

*Top-down tree transducers* (DTTs) are able to produce transformations that depend on a top-down flow of information. To do this, the rules of a DTT specify, for each function symbol $f$, (1) how a state is propagated from a node (labelled with $f$) to its successor nodes, and (2) a tree that replaces the node (labelled with $f$). More formally, a (deterministic and total) DTT from signature $\mathcal{F}$ to signature $\mathcal{G}$ consists of a finite set of states $Q$, an initial state $q_0 \in Q$ and a set of transduction rules of the form

$$q(f(x_1, \ldots, x_n)) \to u \quad \text{for each } f/n \in \mathcal{F} \text{ and } q \in Q$$

where the right-hand side $u \in \mathcal{T}(\mathcal{G}, Q(\mathcal{X}))$ is a term over $\mathcal{G}$ and $Q(\mathcal{X}) = \{p(x_i) \mid p \in Q, 1 \leq i \leq n\}$. That is, the right-hand side is a term that may have subterms of the form $p(x_i)$ with $x_i$ a variable from the left-hand side and $p$ a state in $Q$. In other words, each *occurrence* of a variable on the right-hand side is given a successor state. We only consider DTTs that are deterministic and total, which means that for each $f \in \mathcal{F}$ and $q \in Q$ there is exactly one rule with left-hand side $q(f(x_1, \ldots, x_n))$. A DTT from $\mathcal{F}$ to $\mathcal{G}$ defines a function $\mathcal{T}(\mathcal{F}) \to \mathcal{T}(\mathcal{G})$.

Figure 1 illustrates the shape of the transduction rules of a DTT. On the left-hand side, we find a node with a function symbol $f \in \mathcal{F}$ annotated by a state $q \in Q$. Furthermore, we find the successor nodes (indicated by orange circles), which are represented by variables $x_i$ in the above notation. Intuitively, these variables are *placeholders*, which are instantiated when the rule is applied. On the right-hand side we find a tree over $\mathcal{G}$ that may also contain placeholders taken from the left-hand side. However, each such placeholder has to be annotated with a successor state.

When applying such a rule, the placeholders are instantiated by the actual successor nodes of an $f$-labelled node in the input tree. The application of a rule at the root of a tree is illustrated in Figure 2. The tree has an $f$-labelled root node annotated with a state $q$, and each of its successor nodes is the root of a tree that represents the corresponding subterm. The rule application replaces the root node with a tree that contains subtrees of the original tree as specified in the rule. After this first application of a transduction rule, the process is repeated recursively at each node annotated with a state. In Figure 2, this is the case for the states $q_1$ and $q_2$.

In order to run a DTT on a term $t \in \mathcal{T}(\mathcal{F})$, we have to provide an initial state $q_0$ and then apply the transduction rules to $q_0(t)$ in a top-down fashion. Eventually, this yields a result term $t' \in \mathcal{T}(\mathcal{G})$.

**Example 1.** Consider the signature $\mathcal{F} = \{\mathsf{or}/2, \mathsf{and}/2, \mathsf{not}/1, \mathsf{tt}/0, \mathsf{ff}/0, \mathsf{b}/0\}$. Terms over $\mathcal{F}$ are supposed to represent Boolean expressions with a single Boolean variable $\mathsf{b}$. We construct a DTT from $\mathcal{F}$ to $\mathcal{F}$ that implements the transformation of Boolean ex-

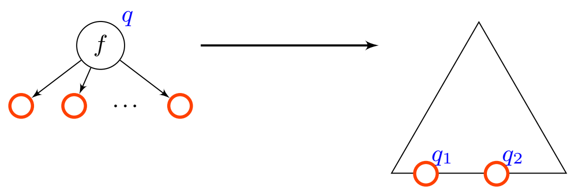


**Figure 1.** Top-down tree transduction rule.

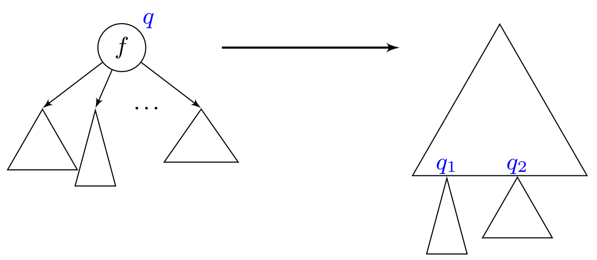


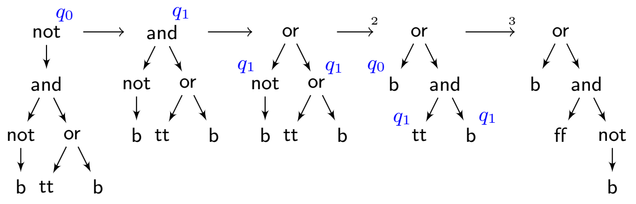
**Figure 2.** Application of top-down tree transduction rule.

pressions into negation normal form, in which negation is only allowed to be applied directly to a variable. The DTT operates on the set of states $Q = \{q_0, q_1\}$ with initial state $q_0$ and the following transduction rules:

$$\begin{aligned}
q_0(\mathsf{and}(x,y)) &\to \mathsf{and}(q_0(x), q_0(y)) & q_0(\mathsf{not}(x)) &\to q_1(x)\\
q_1(\mathsf{and}(x,y)) &\to \mathsf{or}(q_1(x), q_1(y)) & q_1(\mathsf{not}(x)) &\to q_0(x)\\
q_0(\mathsf{or}(x,y)) &\to \mathsf{or}(q_0(x), q_0(y)) & q_0(\mathsf{b}) &\to \mathsf{b}\\
q_1(\mathsf{or}(x,y)) &\to \mathsf{and}(q_1(x), q_1(y)) & q_1(\mathsf{b}) &\to \mathsf{not}(\mathsf{b})
\end{aligned}$$

$$q_0(\mathsf{tt}) \to \mathsf{tt} \quad q_0(\mathsf{ff}) \to \mathsf{ff} \quad q_1(\mathsf{tt}) \to \mathsf{ff} \quad q_1(\mathsf{ff}) \to \mathsf{tt}$$

The transformation that the above DTT performs is straightforward: it moves the operator not inwards, applying De Morgan's laws when it encounters conjunctions and disjunctions. For instance, applied to the expression $\mathsf{not}(\mathsf{and}(\mathsf{not}(\mathsf{b}), \mathsf{or}(\mathsf{tt}, \mathsf{b})))$, the automaton yields the following derivation (where superscripts on the arrows indicate the number of transition steps that have been performed simultaneously):



Note that while states appear as function symbols in the transduction rules we have indicated the states rather as annotations in the derivation above.

In order to start the run of a DTT, the initial state $q_0$ has to be explicitly inserted at the root of the input term. The run of the automaton is completed as soon as all states in the term have vanished; there is no final state.

## 2.3 Terms in Haskell

In Haskell, we represent signatures as (regular) functors. For instance the signature $\mathcal{F}$ from Example 1 is represented in Haskell as follows:

$$\mathbf{data}\ F\ a = \mathit{Or}\ a\ a \mid \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF} \mid B$$

Terms over $F$ are then represented as the free monad $F^*$ of $F$, which is defined as follows:

$$\mathbf{data}\ f^*\ a = \mathit{Re}\ a \mid \mathit{In}\ (f\ (f^*\ a))$$

That is, given a functor $F$ that represents the signature $\mathcal{F}$ and a type $X$ that represents the set $\mathcal{X}$, we use the type $F^*\ X$ to represent the set of terms over $\mathcal{F}$ and $\mathcal{X}$.

For each functor $f$, the type $f^*$ comes with the following generic fold operation:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Alg}\ f\ a = f\ a \to a\\
&\mathit{fold}_*:: \mathit{Functor}\ f \Rightarrow (a \to c) \to \mathit{Alg}\ f\ c \to f^*\ a \to c\\
&\mathit{fold}_*\ \mathit{ret}\ \mathit{alg}\ (\mathit{In}\ t)\ \ = \mathit{alg}\ (\mathit{fmap}\ (\mathit{fold}_*\ \mathit{ret}\ \mathit{alg})\ t)\\
&\mathit{fold}_*\ \mathit{ret}\ \_\ \ \ \ (\mathit{Re}\ x) = \mathit{ret}\ x
\end{aligned}$$

We shall make use of the fact that $f^*$ indeed forms a monad:

$$\begin{aligned}
&\mathbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (f^*)\ \mathbf{where}\\
&\quad \mathit{fmap}\ f = \mathit{fold}_*\ (\mathit{Re} \circ f)\ \mathit{In}\\
&\mathbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (f^*)\ \mathbf{where}\\
&\quad \mathit{return}\ \ = \mathit{Re}\\
&\quad m \gg\!= f = \mathit{fold}_*\ f\ \mathit{In}\ m
\end{aligned}$$

The bind operation of this monad ($\gg\!=$) implements a substitution operation: $t \gg\!= f$ is obtained from $t$ by replacing each subterm $\mathit{Re}\ x$ in $t$ by $f\ x$.

Note that, while we use Haskell as an implementation language, we assume a set-theoretic semantics. That is, all functions are assumed to be total. This assumption is important since we are interested in representing *total* tree transducers only. The full importance of the set theoretic semantics will become apparent when we describe the compositionality of transducers since the compositionality results of (top-down) tree transducers do not hold for tree transducers that are not total [30].

The set theoretic semantics allows us to derive the representation of the set $\mathcal{T}(\mathcal{F})$ from the free monad using an empty type:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Empty}\\
&\mathbf{type}\ \mu f = f^*\ \mathit{Empty}
\end{aligned}$$

The type $\mathit{Empty}$ comes with a canonical mapping $\mathit{empty} :: \mathit{Empty} \to a$, which allows us to derive the fold operation on $\mu f$:

$$\begin{aligned}
&\mathit{fold}_\mu :: \mathit{Functor}\ f \Rightarrow \mathit{Alg}\ f\ c \to \mu f \to c\\
&\mathit{fold}_\mu = \mathit{fold}_*\ \mathit{empty}
\end{aligned}$$

We may use the fold to obtain an embedding of $\mu f$ into $f^*\ a$:

$$\begin{aligned}
&\mathit{toFree} :: \mathit{Functor}\ f \Rightarrow \mu f \to f^*\ a\\
&\mathit{toFree} = \mathit{fold}_\mu\ \mathit{In}
\end{aligned}$$

## 2.4 Top-Down Transduction Functions

We now turn to the representation of DTTs in Haskell. As we have seen in Section 2.2, the transduction rules of a DTT use placeholder variables $x_1, x_2$, etc. in order to refer to arguments of function symbols. These placeholder variables can then be used on the right-hand side of a transduction rule. This mechanism makes it possible to rearrange, remove and duplicate the terms that are matched against these placeholder variables. On the other hand, it is not possible to inspect them. For instance, in Example 1, $q_0(\mathsf{not}(\mathsf{ff})) \to \mathsf{tt}$ would not be a valid transduction rule as we are not allowed to pattern match on the argument of not.

When representing transduction rules as Haskell functions, we have to be careful in order to maintain the restriction described above. In their categorical representation, Hasuo et al. [15] recognised that the restriction due to placeholder variables in the transduction rules can be enforced by a *naturality* condition. Naturality, in turn, can be represented in Haskell's type system as *parametric* *polymorphism*. Following this approach, we represent DTTs from signature functor $f$ to signature functor $g$ with state space $q$ by the following type:

$$\textbf{type}\ \mathit{Trans}_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ (q, f\ a) \to g^{*}\ (q, a)$$

In the definition of tree automata, states are used syntactically as a unary function symbol; a placeholder variable $x$ with state $q$ is written as $q(x)$. In the Haskell representation, we use pairs and simply write $(q, x)$.

In the type $\mathit{Trans}_{\mathsf{D}}$, the type variable $a$ represents the type of the placeholder variables. The universal quantification over $a$ makes sure that these placeholders cannot be scrutinised; they can only be taken from the left-hand side and inserted on the right-hand side without alteration.

**Example 2.** The representation of the signature $\mathcal{F}$ and the state space $Q$ from Example 1 is straightforward:

$$\textbf{data}\ F\ a = \mathit{Or}\ a\ a \mid \mathit{And}\ a\ a \mid \mathit{Not}\ a \mid \mathit{TT} \mid \mathit{FF} \mid B$$
$$\textbf{data}\ Q = \mathit{Q0} \mid \mathit{Q1}$$

For the definition of the transduction function, we use smart constructors $\mathit{iAnd}$, $\mathit{iNot}$, $\mathit{iTT}$, $\mathit{iFF}$ and $\mathit{iB}$ for the constructors of the signature $F$. These smart constructors additionally apply the constructor $\mathit{In}$ of the free monad type. For example:

$$\begin{aligned}&\mathit{iAnd} :: F^{*}\ a \to F^{*}\ a \to F^{*}\ a\\&\mathit{iAnd}\ x\ y = \mathit{In}\ (\mathit{And}\ x\ y)\end{aligned}$$

The transduction function is then defined as follows:

$$\begin{aligned}
&\mathit{trans} :: \mathit{Trans}_{\mathsf{D}}\ F\ Q\ F\\
&\mathit{trans}\ (\mathit{Q0}, \mathit{TT}) = \mathit{iTT}; \qquad \mathit{trans}\ (\mathit{Q0}, \mathit{FF}) = \mathit{iFF}\\
&\mathit{trans}\ (\mathit{Q1}, \mathit{TT}) = \mathit{iFF}; \qquad \mathit{trans}\ (\mathit{Q1}, \mathit{FF}) = \mathit{iTT}\\
&\mathit{trans}\ (\mathit{Q0}, B) = \mathit{iB}\\
&\mathit{trans}\ (\mathit{Q1}, B) = \mathit{iNot}\ \mathit{iB}\\
&\mathit{trans}\ (\mathit{Q0}, \mathit{Not}\ x) = \mathit{Re}\ (\mathit{Q1}, x)\\
&\mathit{trans}\ (\mathit{Q1}, \mathit{Not}\ x) = \mathit{Re}\ (\mathit{Q0}, x)\\
&\mathit{trans}\ (\mathit{Q0}, \mathit{And}\ x\ y) = \mathit{Re}\ (\mathit{Q0}, x)\ \text{`}\mathit{iAnd}\text{`}\ \mathit{Re}\ (\mathit{Q0}, y)\\
&\mathit{trans}\ (\mathit{Q1}, \mathit{And}\ x\ y) = \mathit{Re}\ (\mathit{Q1}, x)\ \text{`}\mathit{iOr}\text{`}\ \ \mathit{Re}\ (\mathit{Q1}, y)\\
&\mathit{trans}\ (\mathit{Q0}, \mathit{Or}\ \ x\ y) = \mathit{Re}\ (\mathit{Q0}, x)\ \text{`}\mathit{iOr}\text{`}\ \ \mathit{Re}\ (\mathit{Q0}, y)\\
&\mathit{trans}\ (\mathit{Q1}, \mathit{Or}\ \ x\ y) = \mathit{Re}\ (\mathit{Q1}, x)\ \text{`}\mathit{iAnd}\text{`}\ \mathit{Re}\ (\mathit{Q1}, y)
\end{aligned}$$

The definition of the transduction function $\mathit{trans}$ is a one-to-one translation of the transduction rules of the DTT from Example 1.

Running a top-down tree transducer on a term is a straightforward affair:

$$\begin{aligned}
&[\![\cdot]\!]_{\mathsf{D}} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow\\
&\quad \mathit{Trans}_{\mathsf{D}}\ f\ q\ g \to q \to \mu f \to \mu g\\
&[\![\mathit{tr}]\!]_{\mathsf{D}}\ q\ t = \mathit{run}\ (q, t)\ \textbf{where}\\
&\quad \mathit{run}\ (q, \mathit{In}\ t) = \mathit{tr}\ (q, t) \gg\!= \mathit{run}
\end{aligned}$$

A top-down transducer is run by applying its transduction function – $\mathit{tr}\ \ (q, t)$ – then recursively running the transformation in the places where the transduction produced placeholders annotated with successor states using the bind of the free monad.

With the thus defined semantics of DTTs in Haskell we can derive the transformation into negation normal form from the DTT given in Example 2:

$$\begin{aligned}&\mathit{negNorm} :: \mu F \to \mu F\\&\mathit{negNorm} = [\![\mathit{trans}]\!]_{\mathsf{D}}\ \mathit{Q0}\end{aligned}$$

### 2.5 Infinite State Space

The Haskell representation of DTTs generalises the definition of DTTs as it does not put any restriction on the state space: the state space $q$ in the type $\mathit{Trans}_{\mathsf{D}}\ f\ q\ g$ does not have to be finite. As a consequence, we can express transformations that go beyond DTTs in the traditional sense, e.g. substitution:

**Example 3.** We consider an expression language with let bindings:

$$\textbf{type}\ \mathit{Var} = \mathit{String}$$
$$\textbf{data}\ \mathit{Sig}\ a = \mathit{Add}\ a\ a \mid \mathit{Val}\ \mathit{Int} \mid \mathit{Let}\ \mathit{Var}\ a\ a \mid \mathit{Var}\ \mathit{Var}$$

Like in Example 2, we assume corresponding smart constructors $\mathit{iAdd}$, $\mathit{iVal}$, $\mathit{iLet}$, and $\mathit{iVar}$.

The DTT in works on a state space of type $\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig}$, i.e. finite maps from variables to terms over $\mathit{Sig}$:

$$\begin{aligned}
&\mathit{trans}_{\mathsf{subst}} :: \mathit{Trans}_{\mathsf{D}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig})\ \mathit{Sig}\\
&\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Var}\ v) \quad\ = \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\ \mathit{Nothing} \to \mathit{iVar}\ v\\
&\qquad\qquad\qquad\qquad\qquad\ \mathit{Just}\ t \quad \to \mathit{toFree}\ t\\
&\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Let}\ v\ b\ s) = \mathit{iLet}\ v\ (\mathit{Re}\ (m, b))\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Re}\ (m \setminus v, s))\\
&\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Val}\ n) \quad\ = \mathit{iVal}\ n\\
&\mathit{trans}_{\mathsf{subst}}\ (m, \mathit{Add}\ x\ y) \ = \mathit{Re}\ (m, x)\ \text{`}\mathit{iAdd}\text{`}\ \mathit{Re}\ (m, y)
\end{aligned}$$

The notation $m \setminus v$ denotes the removal of the mapping for $v$ from $m$. We thus obtain the substitution function as follows:

$$\begin{aligned}&\mathit{subst} :: \mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig} \to \mu\mathit{Sig} \to \mu\mathit{Sig}\\&\mathit{subst} = [\![\mathit{trans}_{\mathsf{subst}}]\!]_{\mathsf{D}}\end{aligned}$$

The initial state for the DTT is simply the substitution that is supposed to be applied to the input term.

The restriction to finite state spaces is necessary to obtain certain decidability results, e.g. the decidability of equivalence for deterministic DTTs [8, 28, 38]. However, our interest in tree transducers here is not the decidability of meta properties but rather the compositionality of tree transducers. In particular, tree transducers are closed under sequential composition [27]. That is, there is an effective procedure that takes two (total, deterministic, top-down) tree transducers $A$ (from $\mathcal{F}$ to $\mathcal{G}$) and $B$ (from $\mathcal{G}$ to $\mathcal{H}$), and produces a tree transducer $C$ (from $\mathcal{F}$ to $\mathcal{H}$) with $[\![C]\!] = [\![B]\!] \circ [\![A]\!]$, i.e. executing $C$ is equivalent to first executing $A$ and then $B$. This construction provides a means to perform program transformations in order to avoid constructing intermediate results, which often results in improved performance [21–23, 34, 35].

These compositionality results make use of the finiteness of the transducers' state spaces in order to construct compositions. We shall see in Section 6, however, that compositionality is maintained even in the case of infinite state spaces. In particular, we show a composition operator

$$\cdot \circ_{\mathsf{D}} \cdot :: \mathit{Trans}_{\mathsf{D}}\ g\ p\ h \to \mathit{Trans}_{\mathsf{D}}\ f\ q\ g \to \mathit{Trans}_{\mathsf{D}}\ f\ (q, p)\ h$$

such that

$$[\![\mathit{tr}_2 \circ_{\mathsf{D}} \mathit{tr}_1]\!]_{\mathsf{D}}\ (q_1, q_2) = [\![\mathit{tr}_2]\!]_{\mathsf{D}}\ q_2 \circ [\![\mathit{tr}_1]\!]_{\mathsf{D}}\ q_1$$

This generalised compositionality also holds for bottom-up tree transducers as well as macro tree transducers, which we shall discuss in Section 3 and 4.

## 3. Tree Transducers with Parametric State Space

The generalised notion of top-down tree transducers that we obtain in Haskell is quite expressive due to the availability of arbitrary state spaces. However, when working with them one quickly realises an unexpected restriction in using these state spaces.

### 3.1 The Problem

Let us reconsider the Haskell DTT from Example 3, which performs substitution. Inspired by this DTT, we would like to define another one that performs inlining of let bindings. This transformation works similarly to the substitution; it only differs in the case of let bindings:

$$
\begin{array}{lll}
trans_{\mathsf{inline}} :: Trans_{\mathsf{D}}\ Sig\ (Map\ Var\ \mu Sig)\ Sig & & \\
trans_{\mathsf{inline}}\ (m, Var\ v) & = \mathbf{case}\ Map.lookup\ v\ m\ \mathbf{of} & \\
 & \quad Nothing \to iVar\ v & \\
 & \quad Just\ e \quad \to toFree\ e & \\
trans_{\mathsf{inline}}\ (m, Let\ v\ x\ y) & = Re\ (m\,[v \mapsto x]\,, y) & \\
trans_{\mathsf{inline}}\ (m, Val\ n) & = iVal\ n & \\
trans_{\mathsf{inline}}\ (m, Add\ x\ y) & = Re\ (m, x)\ `iAdd`\ Re\ (m, y) &
\end{array}
$$

In the case of $Let$, the DTT simply returns the subterm that is in the scope of the let binding, viz. $y$, and updates the substitution to include the mapping $v \mapsto x$. The problem is, however, that this definition does not work as it is not well-typed. The right-hand side $Re\ (m\,[v \mapsto x]\,, y)$ of the offending equation has type $Free\ Sig\ (Map\ Var\ a, a)$ instead of the required type $Free\ Sig\ (Map\ Var\ \mu Sig, a)$. The problem is that the Haskell variable $x$, which we want to put into the substitution, is of type $a$.

Let us recall the type of DTTs in Haskell:

$$\mathbf{type}\ Trans_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ (q, f\ a) \to g^{*}\ (q, a)$$

This type was designed such that placeholder variables like the abovementioned variable $x$ can only be put into the free monad structure on the right-hand side, guarded by a successor state. In particular, we cannot put such a placeholder variable 'into' a state, which means that it is not possible to store subterms – such as the right-hand side of a let binding – into the state and retrieve it later.

## 3.2 The Solution

In order to allow placeholder variables to be put into the state we have to use a state space that is parametric, i.e. not a type but a type function of kind $* \to *$:

$$\mathbf{type}\ Trans\ f\ q\ g = \forall\ a\ .\ (q\ a, f\ a) \to g^{*}\ (q\ a, a)$$

Using this type, the definition of $trans_{\mathsf{inline}}$ is well-typed – with $q = Map\ Var$. However, this type breaks one of the key properties that we have for DTTs, namely that each placeholder variable $x$ that occurs on the right-hand side of a transduction rule is guarded by a state $q$, which is represented in Haskell as a pair $(q, x)$. We do not have this property in the above type. In particular, the placeholder variables that might be used to construct a value of type $q\ a$ are not guarded by a state themselves.

We could address this issue by using a recursive type $P$ that makes sure that all placeholder variables are guarded by a state:

$$\mathbf{data}\ P\ q\ a = P\ (q\ (P\ q\ a))\ a$$

We could then use $P$ to define the type for transducers with a parametric state space:

$$\mathbf{type}\ Trans\ f\ q\ g = \forall\ a\ .\ (q\ a, f\ a) \to g^{*}\ (P\ q\ a)$$

But even this type would not work out right. The problem is that now not only placeholder variables stemming from $f\ a$ have to be guarded by a state but also placeholder variables that have been extracted from the state of type $q\ a$. We want to avoid the latter, since the terms put into the states already have been transformed; thus there is no need to give them a successor state. This problem could be solved by introducing a second type variable $b$ in order to distinguish the two types of placeholder variables. But at this point the resulting type would become much too complicated.

Instead, we will use the type variable $a$ to encode placeholder variables that need not be guarded by a state and the function type $q\ a \to a$ for those that do need to be guarded by a state:

$$\mathbf{type}\ Trans\ f\ q\ g = \forall\ a\ .\ q\ a \to f\ (q\ a \to a) \to g^{*}\ a$$

This encoding also allows us to get rid of the type $P$. Moreover, note that we changed to a curried style instead of using a pair as the first argument. The pair type was used before only in order to stress the representation of state annotations by pairing, which we are about to change now.

We can further generalise this type: as it stands, we can only put placeholder variables into states, which is overly restrictive. In general we should be able to put in arbitrary terms that may contain placeholder variables. We thus arrive at the following type:

$$\mathbf{type}\ Trans_{\mathsf{M}}\ f\ q\ g = \forall\ a\ .\ q\ a \to f\ (q\ (g^{*}\ a) \to a) \to g^{*}\ a$$

Before defining the semantics of this transducer, which will turn out to represent *macro tree transducers*, we shall look at an example implementing the inlining transformation that we started with in this section:

$$
\begin{array}{ll}
trans_{\mathsf{inline}} :: Trans_{\mathsf{M}}\ Sig\ (Map\ Var)\ Sig & \\
trans_{\mathsf{inline}}\ m\ (Var\ v) & = \mathbf{case}\ Map.lookup\ v\ m\ \mathbf{of} \\
 & \quad Nothing \to iVar\ v \\
 & \quad Just\ e \quad \to Re\ e \\
trans_{\mathsf{inline}}\ m\ (Let\ v\ x\ y) & = Re\ (y\ (m'\,[v \mapsto Re\ (x\ m')])) \\
\quad \mathbf{where}\ m' = fmap\ Re\ m & \\
trans_{\mathsf{inline}}\ m\ (Val\ n) & = iVal\ n \\
trans_{\mathsf{inline}}\ m\ (Add\ x\ y) & = Re\ (x\ m')\ `iAdd`\ Re\ (y\ m') \\
\quad \mathbf{where}\ m' = fmap\ Re\ m &
\end{array}
$$

The first change that we can observe in the above code is that occurrences of the placeholder variables on the right-hand side are assigned a state *not* by pairing them with a state – $(m', x)$ – but by applying them to a state – $(x\ m')$.

The implementation of $trans_{\mathsf{inline}}$ illustrates the power of the $Trans_{\mathsf{M}}$ recursion scheme in the case for $Let$: first it passes the current state $m'$ unaltered to $x$, which is then used to add a variable assignment to the state $m'$, which is then passed to $y$. In general, this nesting of recursion can be arbitrarily deep. The type $Trans_{\mathsf{M}}$ encodes this arbitrarily deep nesting of recursion by using the type $q\ (g^{*}\ a) \to a$ for its input placeholder variables.

## 3.3 A More Concise Notation

While we can express the inline transformation with this transducer, it is quite tedious to do so. Since we have to inject the placeholder variables (of type $a$) from the left-hand side into the term structure of the right-hand side (type $Sig^{*}\ a$), the code is riddled with applications of $Re$. However, the encoding of state propagation affords us the opportunity to avoid this explicit plumbing: the only thing we can do with placeholder variables of type $a$ is to inject them into the type $Sig^{*}\ a$ via $Re$. We can build this into the recursion scheme, which gives us the following variant of $Trans_{\mathsf{M}}$:

$$
\begin{array}{l}
\mathbf{type}\ Trans'_{\mathsf{M}}\ f\ q\ g = \forall\ a\ .\ q\ (g^{*}\ a) \to \\
\qquad\qquad f\ (q\ (g^{*}\ a) \to g^{*}\ a) \to g^{*}\ a
\end{array}
$$

We replaced the two "naked" occurrences of the type variable $a$ by $g^{*}\ a$. The following function translates transducers of this more convenient type into the original $Trans_{\mathsf{M}}$ type by applying $Re$ at the appropriate places:

$$
\begin{array}{l}
\Uparrow_{\mathsf{M}} :: (Functor\ f, Functor\ q) \Rightarrow \\
\qquad Trans'_{\mathsf{M}}\ f\ q\ g \to Trans_{\mathsf{M}}\ f\ q\ g \\
\Uparrow_{\mathsf{M}}\ tr\ q\ t = tr\ (fmap\ Re\ q)\ (fmap\ (Re\ \circ)\ t)
\end{array}
$$

With this more convenient type, the inline transformation is implemented without syntactic noise:

$$
\begin{array}{ll}
trans_{\mathsf{inline}} :: Trans'_{\mathsf{M}}\ Sig\ (Map\ Var)\ Sig & \\
trans_{\mathsf{inline}}\ m\ (Var\ v) & = \mathbf{case}\ Map.lookup\ v\ m\ \mathbf{of} \\
 & \quad Nothing \to iVar\ v \\
 & \quad Just\ e \to e
\end{array}
$$

$$\begin{aligned}
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Let}\ v\ x\ y) = y\ (m\,[v \mapsto x\ m])\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Val}\ n) \quad\ = i\mathit{Val}\ n\\
&\mathit{trans}_{\mathsf{inline}}\ m\ (\mathit{Add}\ x\ y) \ = x\ m\ \text{‘}i\mathit{Add}\text{‘}\ y\ m
\end{aligned}$$

## 3.4 The Semantics of $\mathit{Trans}_{\mathsf{M}}$

To better understand the semantics of $\mathit{Trans}_{\mathsf{M}}$, we shall reconsider transducers of type $\mathit{Trans}_{\mathsf{D}}$ first. But we alter the type slightly such that it uses the same style of passing along states as $\mathit{Trans}_{\mathsf{M}}$, i.e. using a function type instead of pairing:

$$\begin{aligned}
&\textbf{type}\ \mathit{Trans}_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ q \to f\ (q \to a) \to g^*\ a\\
&[\![\cdot]\!]_{\mathsf{D}} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow\\
&\quad \mathit{Trans}_{\mathsf{D}}\ f\ q\ g \to q \to \mu f \to \mu g\\
&[\![\mathit{tr}]\!]_{\mathsf{D}}\ \ q\ t = \mathit{run}\ t\ q\ \textbf{where}\\
&\quad \mathit{run}\ (\mathit{In}\ t)\ q = \mathit{join}\ (\mathit{tr}\ q\ (\mathit{fmap}\ \mathit{run}\ t))
\end{aligned}$$

There are no essential differences between this type and the old one; one can give a semantics preserving bijection between the two.

In the above definition, *join* is the multiplication operation of monads, which can be derived from the bind operation:

$$\mathit{join}\ x = x \ggg \mathit{id}$$

Specialised to the free monad constructor, its type is

$$\mathit{join} :: \mathit{Functor}\ f \Rightarrow f^*\ (f^*\ a) \to f^*\ a$$

We finally turn to the semantics of $\mathit{Trans}_{\mathsf{M}}$.

$$\begin{aligned}
&[\![\cdot]\!]_{\mathsf{M}} :: (\mathit{Functor}\ g, \mathit{Functor}\ f, \mathit{Functor}\ q) \Rightarrow\\
&\quad \mathit{Trans}_{\mathsf{M}}\ f\ q\ g \to q\ \mu g \to \mu f \to \mu g\\
&[\![\mathit{tr}]\!]_{\mathsf{M}}\ \ q\ t = \mathit{run}\ t\ q\ \textbf{where}\\
&\quad \mathit{run}\ (\mathit{In}\ t)\ q = \mathit{join}\ (\mathit{tr}\ q\ (\mathit{fmap}\ \mathit{run}'\ t))\\
&\quad \mathit{run}'\ t \qquad\ \ q = \mathit{run}\ t\ (\mathit{fmap}\ \mathit{join}\ q)
\end{aligned}$$

The semantics of $\mathit{Trans}_{\mathsf{M}}$ differs from the definition of the DTT semantics $[\![\cdot]\!]_{\mathsf{D}}$ only slightly. The only significant difference is the use of the auxiliary function $\mathit{run}'$ in order to apply *fmap join* to the states. The necessity of this can be observed in the type definition of $\mathit{Trans}_{\mathsf{M}}$, where the states of type $q\ a$ are transformed into states of type $q\ (g^*\ a)$ before being assigned to a placeholder variable. In the instantiation of this type that is used in the definition of $[\![\cdot]\!]_{\mathsf{M}}$, this means that $q\ \mu g$ is transformed into $q\ (g^*\ \mu g)$. In order to use these states of type $q\ (g^*\ \mu g)$ recursively, they have to be turned into states of type $q\ \mu g$, which is exactly what *fmap join* does.

With this semantics we can finally define the inlining transformation, giving the empty mapping $\emptyset$ as the initial state:

$$\begin{aligned}
&\mathit{inline} :: \mu \mathit{Sig} \to \mu \mathit{Sig}\\
&\mathit{inline} = [\![\Uparrow_{\mathsf{M}}\ \ \mathit{trans}_{\mathsf{inline}}]\!]_{\mathsf{M}}\ \emptyset
\end{aligned}$$

As a final remark we note that the new type for $\mathit{Trans}_{\mathsf{D}}$ allows us to avoid the use of explicit *Re* applications for DTTs as well:

$$\begin{aligned}
&\textbf{type}\ \mathit{Trans}'_{\mathsf{D}}\ f\ q\ g = \forall\ a\ .\ q \to f\ (q \to g^*\ a) \to g^*\ a\\
&\Uparrow_{\mathsf{D}}\ ::\mathit{Functor}\ f \Rightarrow \mathit{Trans}'_{\mathsf{D}}\ f\ q\ g \to \mathit{Trans}_{\mathsf{D}}\ f\ q\ g\\
&\Uparrow_{\mathsf{D}}\ \ \mathit{tr}\ q\ t = \mathit{tr}\ q\ (\mathit{fmap}\ (\mathit{Re} \circ)\ t)
\end{aligned}$$

For example, the definition of the substitution transducer now looks like this:

$$\begin{aligned}
&\mathit{trans}_{\mathsf{subst}} :: \mathit{Trans}'_{\mathsf{D}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var}\ \mu\mathit{Sig})\ \mathit{Sig}\\
&\mathit{trans}_{\mathsf{subst}}\ m\ (\mathit{Var}\ v) \qquad = \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Nothing} \to i\mathit{Var}\ v\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Just}\ t \to \mathit{toFree}\ t\\
&\mathit{trans}_{\mathsf{subst}}\ m\ (\mathit{Let}\ v\ b\ s) = i\mathit{Let}\ v\ (b\ m)\ (s\ (m \setminus v))\\
&\mathit{trans}_{\mathsf{subst}}\ m\ (\mathit{Val}\ n) \qquad = i\mathit{Val}\ n\\
&\mathit{trans}_{\mathsf{subst}}\ m\ (\mathit{Add}\ x\ y) \ \ = x\ m\ \text{‘}i\mathit{Add}\text{‘}\ y\ m
\end{aligned}$$

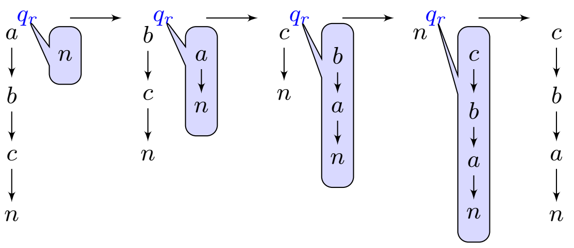


**Figure 3.** Run of an MTT.

$$\begin{aligned}
&\mathit{subst} :: \mathit{Map}\ \ \mathit{Var}\ \mu\mathit{Sig} \to \mu\mathit{Sig} \to \mu\mathit{Sig}\\
&\mathit{subst} = [\![\Uparrow_{\mathsf{D}}\ \ \mathit{trans}_{\mathsf{subst}}]\!]_{\mathsf{D}}
\end{aligned}$$

In the following section we shall see that the transducer type $\mathit{Trans}_{\mathsf{M}}$ represents a well-studied extension of top-down (and bottom-up) tree transducers, namely macro tree transducers.

## 4. Macro Tree Transducers

Macro tree transducers (MTTs) [9] generalise DTTs by adding accumulation parameters to states. A simple example of using an accumulation parameter is the implementation of the reversal of lists. For simplicity we encode lists as terms over the signature $\mathcal{F} = \{a/1, b/1, c/1, n/0\}$, where $a$, $b$ and $c$ are the list elements and $n$ is the empty list. For example the list containing the elements $a$, $b$ and $c$ is represented by the term $a(b(c(n)))$.

**Example 4.** The reversal transformation is implemented using a single state $q_r$ with a single accumulation parameter. Accumulation parameters are added to states by simply increasing their arities. In DTTs, states are represented by unary function symbols. For MTTs, we increment the arity for each additional accumulation parameter. In the case of the reversal transformation, $q_r$ has arity two. The elements of the list are put into the accumulator one by one, and once the end of the original list is reached, the accumulator holds the reversal of the list.

$$\begin{aligned}
q_r(a(x), y) &\to q_r(x, a(y))\\
q_r(b(x), y) &\to q_r(x, b(y))\\
q_r(c(x), y) &\to q_r(x, c(y))\\
q_r(n, y) &\to y
\end{aligned}$$

The semantics of MTTs is similar to the semantics of DTTs: simply apply the rules as rewrite rules. For example starting with the term $a(b(c(n)))$, state $q_r$ and accumulator $n$, we obtain the following run:

$$\begin{aligned}
q_r(a(b(c(n))), n) &\to q_r(b(c(n)), a(n)) \to q_r(c(n), b(a(n)))\\
&\to q_r(n, c(b(a(n)))) \to c(b(a(n))).
\end{aligned}$$

A graphical representation of this run is given in Figure 3. Again we depict the states as annotations rather than nodes in the tree. The difference compared to DTTs is that the each state has a fixed number of memory slots (viz. the accumulation parameters) in which to store trees. In this example, the state $q_r$ has a single memory slot, which is depicted as a blue box in the figure. The result of running the transducer on $a(b(c(n)))$ is the reversal $c(b(a(n)))$.

In general, a macro tree transducer (MTT) from signature $\mathcal{F}$ to signature $\mathcal{G}$ consist of a signature $Q$ of states with arity $> 0$, and a set of transduction rules of the form

$$q(f(x_1, \ldots, x_n), y_1, \ldots, y_m) \to u \qquad \begin{array}{r}\text{for each } f/n \in \mathcal{F}\\ \text{and } q/(m+1) \in Q\end{array}$$

where $u$ is an element of the set $RHS_{n,m}$, which is inductively defined by the formation rules in Figure 4. In short, the right-hand side

$$\frac{1 \le i \le m}{y_i \in RHS_{n,m}} \qquad \frac{g/k \in \mathcal{G} \quad u_1, \ldots, u_k \in RHS_{n,m}}{g(u_1, \ldots, u_k) \in RHS_{n,m}}$$

$$\frac{1 \le i \le n \quad p/(k+1) \in Q \quad u_1, \ldots, u_k \in RHS_{n,m}}{p(x_i, u_1, \ldots, u_k) \in RHS_{n,m}}$$

**Figure 4.** Inductive definition of $RHS_{n,m}$.

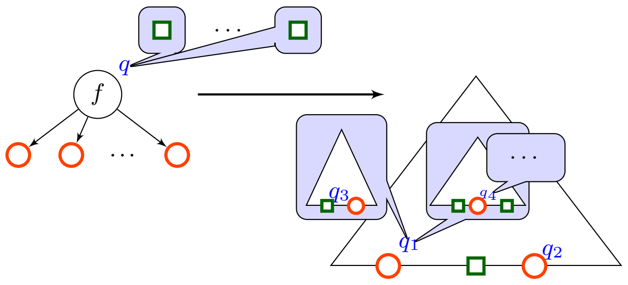


**Figure 5.** Macro tree transduction rule.

may contain *accumulation variables* $y_i$, a function symbol application $g(u_1, \ldots, u_k)$ or *input variables* $x_i$ guarded by a successor state $p$ with appropriate accumulation arguments $u_1, \ldots u_k$.

The shape of the transduction rules of MTTs is depicted in Figure 5. It is instructive to compare this depiction with the depiction of DTT transduction rules in Figure 1. The basic shape of a MTT transduction rule is similar to the DTT transduction rules. The only aspect that changes is that states have more structure: each state has a fixed number of "memory cells" in which accumulation parameters can be stored and retrieved. Accumulation variables are indicated by green squares. They appear in the state on the left-hand side and can be used in building trees on the right-hand side. The difference compared to the input variables (indicated by orange circles as for DTTs) is that accumulation variables are not guarded by a successor state. Moreover, each state on the right-hand side may also have a number of "memory cells", which must be filled with trees as well. These trees have the same structure again, containing input variables guarded by states as well as accumulation variables. This nesting of trees can be arbitrarily deep, thus allowing an arbitrarily deeply nested recursion.

Similarly to DTT transduction rules, when applying MTT transduction rules, the placeholder variables (both the input variables and the accumulation variables) are instantiated with concrete trees.

With the type $Trans_{\mathsf{M}}$ we are able to represent MTTs. As for DTTs, the signatures $\mathcal{F}$ and $\mathcal{G}$ are represented as Haskell functors. In the same way also the state space is represented as a functor.

$$\textbf{data}\ F\ a = A\ a \mid B\ a \mid C\ a \mid N$$
$$\textbf{data}\ Q\ a = Qr\ a$$

The representation of the input signature is straightforward. Since the original MTT only has one state $q$, the type $Q$ also has only one constructor. The constructor $Qr$ has one argument as the state $q_r$ in the original MTT has one accumulation parameter.

For the definition of the MTT in Haskell we again assume appropriate smart constructors for the target signature; in this case $iA$, $iB$, $iC$ and $iN$:

$$\begin{aligned}
&trans_{\mathsf{rev}} :: Trans'_{\mathsf{M}}\ F\ Q\ F\\
&trans_{\mathsf{rev}}\ (Qr\ y)\ (A\ x) = x\ (Qr\ (iA\ y))\\
&trans_{\mathsf{rev}}\ (Qr\ y)\ (B\ x) = x\ (Qr\ (iB\ y))\\
&trans_{\mathsf{rev}}\ (Qr\ y)\ (C\ x) = x\ (Qr\ (iC\ y))\\
&trans_{\mathsf{rev}}\ (Qr\ y)\ N \qquad = y
\end{aligned}$$

The above definition is a one-to-one translation of the transduction rules of the original MTT. We thus obtain the implementation of *reverse* as follows:

$$\begin{aligned}
&reverse :: \mu F \to \mu F\\
&reverse = [\![\Uparrow_{\mathsf{M}}\ \ trans_{\mathsf{rev}}]\!]_{\mathsf{M}}\ (Qr\ iN)
\end{aligned}$$

The MTT given in Example 4 illustrates that states in MTTs correspond to recursively defined functions. The state $q_r$ is a function that performs the reversal of lists, with the help of an additional accumulation parameter. Multiple states, then correspond to several mutually recursively defined functions, e.g. the two states $q_0$ and $q_1$ in Example 1 correspond to two mutually recursive functions. In our Haskell representation, however, we view the states of MTTs really as states. Hence, potentially infinite state spaces arise naturally. This aspect differentiates our work from the typical view of tree transducers as (restricted) functional programs [21–23].

## 5. Annotating Trees

Before discussing the aspect of compositionality of MTTs, we give a brief example that illustrates the flexibility that the use of MTTs affords us.

When working with abstract syntax trees (AST), e.g. in a compiler, it is common that the that the AST produced by the parser has addition annotations attached to each node, e.g. the location in a source file from which the node originated. This information can then be used for giving useful error and warning messages.

We can easily extend a functor $f$ to a functor $an$ :&: $f$ that contains annotations of type $an$:

$$\textbf{data}\ (an :\&: f)\ a = an :\&: (f\ a)$$

Trees of type $\mu(an :\&: f)$ are like trees of type $\mu f$, but additionally each node is annotated with an annotation of type $an$.

However, many operations on ASTs typically ignore the annotation information as it is not needed. The structure of transducers makes it quite easy to lift them from an arbitrary source signature to a source signature with annotations.

$$\begin{aligned}
&ignoreAnn :: (Functor\ g, Functor\ q, Functor\ f) \Rightarrow\\
&\quad Trans_{\mathsf{M}}\ f\ q\ g \to Trans_{\mathsf{M}}\ (an :\&: f)\ q\ g\\
&ignoreAnn\ tr\ q\ (\_ :\&: t) = tr\ q\ t
\end{aligned}$$

In other cases, the annotations should be in principle ignored but nevertheless propagated to the result. For example, it we want to use our inlining transformation from Section 3.3 in a compiler we may want to extend it such that annotations are preserved. We have shown previously [2] that such a lifting can be performed generically for rather simple transformations known as tree homomorphisms[1], which for example occur in the form of *desugaring* transformations. The same idea can also be applied to MTTs. To this end, we need a function that adds a given annotation to every node in a tree:

$$\begin{aligned}
&addAnn :: Functor\ f \Rightarrow an \to f^*\ a \to (an :\&: f)^*\ a\\
&addAnn\ an\ (In\ t) = In\ (an :\&: fmap\ (addAnn\ an)\ t)\\
&addAnn\ \_\ (Re\ x) = Re\ x
\end{aligned}$$

This transformation can in fact be described as a DTTs with a singleton state space. We can then use this function to construct the lifting of MTTs such that they propagate annotations:

$$\begin{aligned}
&propAnn :: (Functor\ g, Functor\ q, Functor\ f) \Rightarrow\\
&\quad Trans_{\mathsf{M}}\ f\ q\ g \to Trans_{\mathsf{M}}\ (an :\&: f)\ q\ (an :\&: g)\\
&propAnn\ tr\ q\ (an :\&: t) = addAnn\ an\\
&\qquad\qquad\qquad\qquad\qquad (tr\ q\ (fmap\ addAnn'\ t))\\
&\quad \textbf{where}\ addAnn'\ f\ q' = f\ (fmap\ (addAnn\ an)\ q')
\end{aligned}$$

[1] Tree homomorphisms are simply DTTs with a singleton state space

As expected we simply use $addAnn$ to annotate the result of the original MTT $tr$ with the annotation $an$. However, due to the natrue of MTTs, we have to also propagate the annotaion to the trees that are put *into* accumulation parameters. This task is achieved by the $addAnn'$ function.

Using the thus defined lifting function, we derive the following variant of the inlining transformation that propagates annotations faithfully:

$$\begin{array}{l} inlineAnnotated :: \mu(an :\&: Sig) \to \mu(an :\&: Sig) \\ inlineAnnotated = [\![ propAnn\ (\Uparrow_{\mathsf{M}}\ \ trans_{\mathsf{inline}}) ]\!]_{\mathsf{M}}\ \emptyset \end{array}$$

We close this section with a small example that shows how to generically annotate a tree with unique labels so as to track changes made by subsequent transformations. In particular, we shall annotate each node with a list of type $[Int]$ that describes its access path, i.e. the path one has to take from the root in order to reach the node. For example, in the tree representing the term $iLet$ `"x"` $(iVal\ 1$ ‘$iAdd$‘ $iVal\ 2)$ $(iVar$ `"x"` ‘$iAdd$‘ $iVar$ `"x"`$)$, the node representing the subterm $iVal\ 2$ has the access path $[0, 1]$. We perform this transformation using the type class $Traversable$, which for each instance $Traversable\ t$ provides the method

$$mapM :: Monad\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)$$

generalising the $mapM$ function on lists to other functors. Instances of $Traversable$ can be derived mechanically in GHC, in particular for the class of regular functors that we consider here. Using this generalised $mapM$ function we can use a state monad to number the arguments of each constructor of a (traversable) functor:

$$\begin{array}{l} number :: Traversable\ f \Rightarrow f\ a \to f\ (Int, a) \\ number\ x = fst\ (runState\ (mapM\ run\ x)\ 0)\ \mathbf{where} \\ \quad run\ b = \mathbf{do}\ n \leftarrow get; put\ (n+1); return\ (n, b) \end{array}$$

For example, applied to $Let\ v\ r\ e$, the function $number$ produces $Let\ v\ (0, r)\ (1, e)$.

Equipped with this tool, we can finally define the DTT that performs the desired transformation:

$$\begin{array}{l} trans_{\mathsf{path}} :: Traversable\ f \Rightarrow Trans_{\mathsf{D}}\ f\ [Int]\ ([Int] :\&: f) \\ trans_{\mathsf{path}}\ q\ t = inFree\ (q :\&: fmap\ (\lambda(n, s) \to s\ (n : q)) \\ \qquad\qquad\qquad\qquad\qquad\qquad (number\ t)) \\ inFree :: Functor\ f \Rightarrow f\ a \to f^*\ a \\ inFree\ t = In\ (fmap\ Re\ t) \end{array}$$

We thus arrive at the following transformation function, which adds path annotations to any tree:

$$\begin{array}{l} pathAnn :: Traversable\ f \Rightarrow \mu f \to \mu([Int] :\&: f) \\ pathAnn = [\![ trans_{\mathsf{path}} ]\!]_{\mathsf{D}}\ [\,] \end{array}$$

In the next section, we shall see how to compose tree transducers, which will allow us to combine $trans_{\mathsf{path}}$ with other transducers in order to track the transformations they perform.

## 6. Composition of Tree Transducers

In this section we demonstrate the compositionality of tree transducers in Haskell. Before discussing the composition of MTTs, we have a look at the composition of DTTs.

### 6.1 Composition of DTTs

The sequential composition of two tree transformations expressed as DTTs is itself expressible as a DTT, which can be effectively constructed from the two original DTTs [27]. The idea behind this construction is quite simple: given two DTTs $\mathcal{A}_1$ (from $\mathcal{F}$ to $\mathcal{G}$) and $\mathcal{A}_2$ (from $\mathcal{G}$ to $\mathcal{H}$) we construct a DTT $\mathcal{A}$ (from $\mathcal{F}$ to $\mathcal{H}$) that performs the composition of $\mathcal{A}_2$ and $\mathcal{A}_1$ by taking all the transduction rules from $\mathcal{A}_1$ and transform their right-hand sides by running the DTT $\mathcal{A}_2$ on them. That is for each rule

$$q_1(f(x_1, \ldots, x_n)) \to u \quad \text{in } \mathcal{A}_1$$

and state $p$ in $\mathcal{A}_2$, we produce the rule

$$(q_1, q_2)(f(x_1, \ldots, x_n)) \to [\![\mathcal{A}_2]\!]\ (q_2, u)$$

Note that the state space of $\mathcal{A}$ is the product $Q_1 \times Q_2$ of the state spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$. The additional state information $q_2$ is used as the initial state for running the transducer $\mathcal{A}_2$ on the right-hand side $u$. Technically, the transducer $\mathcal{A}_2$ has to be modified slightly since the term $u$ may contain states from $\mathcal{A}$. This is done by adding rules of the form

$$q_2(q_1(x)) \to (q_1, q_2)(x)$$

to $\mathcal{A}_2$, which simply combine the states from the two transducers to get a state in the compound state space $Q_1 \times Q_2$.

The construction in Haskell follows the same idea. In order to simplify the presentation and the subsequent correctness proof we shall reformulate the definition of the DTT semantics $[\![\cdot]\!]_{\mathsf{D}}$ as a fold of an algebra constructed as follows:

$$\begin{array}{l} alg_{\mathsf{D}} :: Functor\ g \Rightarrow Trans_{\mathsf{D}}\ f\ q\ g \to Alg\ f\ (q \to g^*\ a) \\ alg_{\mathsf{D}}\ tr\ t\ q = join\ (tr\ q\ t) \\ [\![\cdot]\!]_{\mathsf{D}} :: (Functor\ f, Functor\ g) \Rightarrow \\ \qquad Trans_{\mathsf{D}}\ f\ q\ g \to q \to \mu f \to \mu g \\ [\![tr]\!]_{\mathsf{D}}\ q\ t = fold_{\mu}\ (alg_{\mathsf{D}}\ tr)\ t\ q \end{array}$$

The function $alg_{\mathsf{D}}$ turns a DTT transduction function into the algebra that describes its semantics. The semantics $[\![\cdot]\!]_{\mathsf{D}}$ is then simply the fold of this algebra. Using the same algebra, we generalise the semantics $[\![\cdot]\!]_{\mathsf{D}}$ in a way that corresponds to the addition of rules of the form $q_2(q_1(x)) \to (q_1, q_2)(x)$ as described above:

$$\begin{array}{l} (\!|\cdot|\!)_{\mathsf{D}} :: (Functor\ f, Functor\ g) \Rightarrow \\ \quad Trans_{\mathsf{D}}\ f\ q\ g \to q \to f^*\ (q \to a) \to g^*\ a \\ (\!|tr|\!)_{\mathsf{D}}\ q\ t = fold_*\ (\lambda a\ q \to Re\ (a\ q))\ (alg_{\mathsf{D}}\ tr)\ t\ q \end{array}$$

This semantics generalises $[\![\cdot]\!]_{\mathsf{D}}$ in the sense that it works on $f^*\ (q \to a)$ not only $\mu f$. In analogy to the automata theoretic construction sketched above, we will use $(\!|\cdot|\!)_{\mathsf{D}}$ in order to run a DTT on the right-hand side of the transduction rules of another DTT. The composition of DTTs is thus constructed as follows:

$$\begin{array}{l} \cdot \circ_{\mathsf{D}} \cdot :: (Functor\ f, Functor\ g, Functor\ h) \Rightarrow \\ \quad Trans_{\mathsf{D}}\ g\ q_2\ h \to Trans_{\mathsf{D}}\ f\ q_1\ g \to Trans_{\mathsf{D}}\ f\ (q_1, q_2)\ h \\ tr_2 \circ_{\mathsf{D}} tr_1\ (q_1, q_2)\ t = (\!|tr_2|\!)_{\mathsf{D}}\ q_2\ (tr_1\ q_1\ (fmap\ curry\ t)) \end{array}$$

The right-hand side of the first DTT $tr_1$ is obtained by applying it to its component $q_1$ of the compound state space as well as the input pattern $t$. Note that we have to apply currying in order to transform the input pattern for the compound DTT, which is of type $f\ ((q_1, q_2) \to a)$, into an input pattern suitable for the first DTT, viz. of type $f\ (q_1 \to q_2 \to a)$. After this application, the second DTT $tr_2$ is run on the thus obtained right-hand side of $tr_1$.

This construction is the same as the automata-theoretic construction as first shown by Rounds [27]. Since the accompanying soundness proof of Rounds does not make use of the finiteness of the state space, it follows that that also our composition construction is sound. That is, we have the following theorem:

**Theorem 1** (Soundness of DTT composition)**.** *For all $tr_1$, $tr_2$, $q_1$, $q_2$, $t$ of appropriate type, we have the following equality:*

$$[\![tr_2]\!]_{\mathsf{D}}\ q_2\ ([\![tr_1]\!]_{\mathsf{D}}\ q_1\ t) = [\![tr_2 \circ_{\mathsf{D}} tr_1]\!]_{\mathsf{D}}\ (q_1, q_2)\ t$$

However, we can give a rather compact direct proof of this soundness theorem without referring the automata-theoretic proof.

*Proof of Theorem 1.* Let $a_1$, $a_2$ and $a$ be the algebras of $tr_1$, $tr_2$ and $tr_2 \circ_{\mathsf{D}} tr_1$ according to $alg_{\mathsf{D}}$. Unfolding the definition of $[\![\cdot]\!]_{\mathsf{D}}$, we can reformulate the claimed equality as follows:

$$f\ (fold_\mu\ a_1\ t)\ (q_1, q_2) = fold_\mu\ a\ t\ (q_1, q_2) \qquad (1)$$

where $f$ is defined by the equation

$$f\ g\ (q_1, q_2) = fold_\mu\ a_2\ (g\ q_1)\ q_2$$

By the fusion law for folds it suffices to prove the following equality instead:

$$\forall\ x\ .\ fold_\mu\ a_2\ (a_1\ x\ q_1)\ q_2 = a\ (fmap\ f\ x)\ (q_1, q_2) \qquad (2)$$

In order to prove this, we take advantage of the observation that *join* commutes with $fold_*$:[2]

**Lemma 1.** *Let $e = \lambda z\ q \to Re\ (z\ q)$ and $b = alg_{\mathsf{D}}\ tr$ for some $tr$. Then the following holds for all $x$ and $q$:*

$$fold_\mu\ b\ (join\ x)\ q = join\ (fold_*\ e\ b\ (fmap\ (fold_\mu\ b)\ x)\ q)$$

The proof of (2) follows:

$$\begin{aligned}
& a\ (fmap\ f\ x)\ (q_1, q_2) \\
= \ & \{\ a = alg_{\mathsf{D}}\ (tr_2 \circ_{\mathsf{D}} tr_1) \text{ and definition of } alg_{\mathsf{D}}\ \} \\
& join\ ((tr_2 \circ_{\mathsf{D}} tr_1)\ (q_1, q_2)\ (fmap\ f\ x)) \\
= \ & \{\ \text{Definition of } \cdot \circ_{\mathsf{D}} \cdot \text{ and } (\!|\cdot|\!)_{\mathsf{D}}\ \} \\
& join\ (fold_*\ e\ a_2\ (tr_1\ q_1\ (fmap\ (curry \circ f)\ x))\ q_2) \\
= \ & \{\ curry \circ f = \lambda g\ q_1\ q_2 \to fold_\mu\ a_2\ (g\ q_1)\ q_2\ \} \\
& join\ (fold_*\ e\ a_2\ (tr_1\ q_1\ (fmap\ (fold_\mu\ a_2 \circ)\ x))\ q_2) \\
= \ & \{\ \text{Parametricity}\ \} \\
& join\ (fold_*\ e\ a_2\ (fmap\ (fold_\mu\ a_2)\ (tr_1\ q_1\ x))\ q_2) \\
= \ & \{\ \text{Lemma 1}\ \} \\
& fold_\mu\ a_2\ (join\ (tr_1\ q_1\ x))\ q_2 \\
= \ & \{\ a_1 = alg_{\mathsf{D}}\ tr_1 \text{ and definition of } alg_{\mathsf{D}}\ \} \\
& fold_\mu\ a_2\ (a_1\ x\ q_1)\ q_2
\end{aligned}$$

The equation labelled "Parametricity" follows from the equality

$$\forall\ g\ q\ .\ fmap\ g \circ tr\ q = tr\ q \circ fmap\ (g\ \circ)$$

which holds for all $tr$ of type $Trans_{\mathsf{D}}\ f\ q\ g$ because of parametricity [36]. □

Note that in the application of the fusion law for folds we make use of the set theoretic semantics. In a CPO semantics, the function $f$ would also need to satisfy a strictness side condition for the fusion law [25]. This strictness condition does in fact not hold for $f$. However, this should be expected since the closure of DTTs under composition is restricted to *total*, deterministic DTTs. Indeed, Thatcher [30] gives a counterexample for the composition with a deterministic DTT that is not total.

## 6.2 Composition of MTTs

Unlike DTTs, MTTs are in general not closed under composition [9]. That is, a transformation that is expressible as the composition of two MTT-expressible transformations cannot always be expressed as a single MTT. However, for a large class of cases the composition of two MTTs does again yield an MTT. In particular, the composition of an MTT with a DTT (and vice versa) can be expressed as a single MTT, which can be effectively constructed [9].

Similar to the case for DTTs described above, we shall reformulate the semantics of MTTs as a fold:

$$\begin{aligned}
& alg_{\mathsf{M}} :: (Functor\ g, Functor\ f, Functor\ q) \Rightarrow \\
& \qquad Trans_{\mathsf{M}}\ f\ q\ g \to Alg\ f\ (q\ (g^*\ a) \to g^*\ a) \\
& alg_{\mathsf{M}}\ tr\ t\ q = join\ (tr\ q\ (fmap\ (\circ\, fmap\ join)\ t)) \\
& [\![\cdot]\!]_{\mathsf{M}} :: (Functor\ g, Functor\ f, Functor\ q) \Rightarrow \\
& \qquad Trans_{\mathsf{M}}\ f\ q\ g \to q\ \mu g \to \mu f \to \mu g \\
& [\![tr]\!]_{\mathsf{M}}\ q\ t = fold_\mu\ (alg_{\mathsf{M}}\ tr)\ t\ q
\end{aligned}$$

We use the algebra to define a generalised semantics as well:

$$\begin{aligned}
& (\!|\cdot|\!)_{\mathsf{M}} :: (Functor\ g, Functor\ f, Functor\ q) \Rightarrow \\
& \quad Trans_{\mathsf{M}}\ f\ q\ g \to q\ (g^*\ a) \to f^*\ (q\ (g^*\ a) \to a) \to g^*\ a \\
& (\!|tr|\!)_{\mathsf{M}}\ q\ t = fold_*\ (\lambda a\ q \to Re\ (a\ q))\ (alg_{\mathsf{M}}\ tr)\ t\ q
\end{aligned}$$

We first consider the simpler case of the composition, viz. a DTT followed by an MTT. To this end we shall use the direct construction given by Kühnemann [22]. This construction follows essentially the same idea as the composition construction for DTTs described in Section 6.1: we take the rules of the first transducer and transform their right-hand sides by running the second transducer on them. The only difference is that since we are dealing with an MTT, the state space of one of the transducers is a functor. Thus we have to replace pairing with the following type constructor:

$$\mathbf{data}\ (q_1 :\&: q_2)\ a = q_1 :\&: (q_2\ a)$$

We have already seen this construction in Section 5 where it was used for adding annotations to trees.

The definition of the composition operator $\cdot \circ_{\mathsf{MD}} \cdot$ then follows the same pattern as $\cdot \circ_{\mathsf{D}} \cdot$:

$$\begin{aligned}
& \cdot \circ_{\mathsf{MD}} \cdot :: (Functor\ f, Functor\ g, Functor\ h, Functor\ p) \Rightarrow \\
& \quad Trans_{\mathsf{M}}\ g\ p\ h \to Trans_{\mathsf{D}}\ f\ q\ g \to Trans_{\mathsf{M}}\ f\ (q :\&: p)\ h \\
& tr_2 \circ_{\mathsf{MD}} tr_1\ (q_1 :\&: q_2)\ t = (\!|tr_2|\!)_{\mathsf{M}}\ (fmap\ Re\ q_2) \\
& \qquad\qquad\qquad\qquad (tr_1\ q_1\ (fmap\ curryF\ t))
\end{aligned}$$

where $curryF$ is a variant of $curry$ defined on the type constructor :&: instead:

$$\begin{aligned}
& curryF :: ((q :\&: p)\ a \to b) \to q \to p\ a \to b \\
& curryF\ f\ q\ p = f\ (q :\&: p)
\end{aligned}$$

Since the construction given by $\cdot \circ_{\mathsf{MD}} \cdot$ follows the corresponding automata-theoretic construction and since the accompanying soundness proof [9] does not use the finiteness of the state space, we can conclude the soundness of our construction:

**Theorem 2** (Soundness of MTT-DTT composition)**.** *For all $tr_1$, $tr_2$, $q_1$, $q_2$, $t$ of appropriate type, we have the following equality:*

$$[\![tr_2 \circ_{\mathsf{MD}} tr_1]\!]_{\mathsf{M}}\ (q_1 :\&: q_2)\ t = [\![tr_2]\!]_{\mathsf{M}}\ q_2\ ([\![tr_1]\!]_{\mathsf{D}}\ q_1\ t)$$

**Example 5.** As a simple example, we combine the inlining MTT with the DTT that adds annotations:

$$\begin{aligned}
& trans :: Trans_{\mathsf{M}}\ Sig\ ([Int] :\&: (Map\ Var))\ ([Int] :\&: Sig) \\
& trans = (propAnn\ (\Uparrow_{\mathsf{M}}\ trans_{\mathsf{inline}})) \circ_{\mathsf{MD}} trans_{\mathsf{path}}
\end{aligned}$$

Here we use $propAnn$ to lift the original inlining MTT to propagate annotations. We thus obtain the following transformation function by supplying the appropriate initial state:

$$\begin{aligned}
& inlineTrack :: \mu Sig \to \mu([Int] :\&: Sig) \\
& inlineTrack = [\![trans]\!]_{\mathsf{M}}\ ([\,] :\&: \emptyset)
\end{aligned}$$

Finally, we consider the composition of an MTT followed by a DTT. This case is slightly more complicated. It is not sufficient to simply run the DTT on the right-hand sides of the rules of the MTT. In the transduction rules of an MTT, the accumulation variables are used directly on the right-hand side without being guarded by a successor state. The reason for this is the fact that the trees that are stored in the accumulation parameters have already been transformed by the MTT. Hence, in order to achieve the same

[2] We omit the proof of this lemma; it can be found in the associated material on the authors' websites.

invariance for the composition of an MTT followed by a DTT, we have to make sure to also run the DTT on the trees in the accumulation parameters.

In the original automata-theoretic construction of Engelfriet and Vogler [9], this nested run of the DTT is achieved by replacing each accumulation parameter of the original MTT with $k$ copies, where $k$ is the number of states in the DTT. The rules of the MTT are then manipulated such that the $i$-th copy of an accumulation parameter contains the original accumulation parameter but transformed by running the DTT on it using the $i$-th state as initial state.

This construction makes direct use of the fact that the state space of the DTT is finite as each state with $n$ accumulation parameters in the MTT is replaced by a state with $n * k$ accumulation parameters. However, we can apply the same construction for our generalised transducers with potentially infinite state spaces. To this end we exploit the fact that our representation of MTTs does not restrict the number of accumulation parameters to be finite. Given the state space $q_1$ of an MTT, we achieve the construction of producing a copy of each accumulation parameter for each state in the state space $q_2$ of the DTT by using the type $Exp\ q_1\ q_2$:

$$\mathbf{type}\ Exp\ q_1\ q_2\ a = q_1\ (q_2 \to a)$$

The nested function type makes sure that there is a copy of each accumulation parameter for each state in $q_2$.

We then obtain the state space of the compound transducer by pairing the above exponentiated state space with the state space $q_2$ of the DTT, which yields the type $q_2$ :&: $Exp\ q_1\ q_2$. We combine this construction in a single definition as follows:

$$\mathbf{data}\ (q_1 :\wedge: q_2)\ a = q_1\ (q_2 \to a) :\wedge: q_2$$

We thus arrive at the following construction of the composition of DTTs and MTTs:

$$\begin{aligned}&\cdot \circ_{\mathsf{DM}} \cdot :: (Functor\ f, Functor\ g, Functor\ h, Functor\ q) \Rightarrow\\ &\quad Trans_{\mathsf{D}}\ g\ p\ h \to Trans_{\mathsf{M}}\ f\ q\ g \to Trans_{\mathsf{M}}\ f\ (q :\wedge: p)\ h\\ &tr_2 \circ_{\mathsf{DM}} tr_1\ (q_1 :\wedge: q_2)\ t =\\ &\quad (\!|tr_2|\!)_{\mathsf{D}}\ q_2\ (tr_1\ (fmap\ (\$)\ q_1)\ (fmap\ (nested\ tr_2)\ t))\end{aligned}$$

In this case, simple currying is not sufficient for preparing the input pattern $t$. As explained above, we have to run the DTT also on the accumulation parameters. The function $nested$ does exactly that:

$$nested\ tr\ f\ q_1\ q_2 = f\ (fmap\ (\lambda s\ p \to (\!|tr|\!)_{\mathsf{D}}\ p\ s)\ q_1 :\wedge: q_2)$$

To achieve this, $nested$ makes use of the fact that we expand the state space using exponentiation. Thus, we can run the DTT on the tree $s$ using the supplied state $p$ as initial state of the run.

Again, we appeal to the automata-theoretic proof of Engelfriet and Vogler [9] to conclude that the above composition operator $\cdot \circ_{\mathsf{DM}} \cdot$ is sound:

**Theorem 3** (Soundness of MTT-DTT composition)**.** *For all $tr_1$, $tr_2$, $q_1$, $q_2$, $t$ of appropriate type, we have the following equality:*

$$[\![tr_2 \circ_{\mathsf{DM}} tr_1]\!]_{\mathsf{M}}\ (q_1' :\wedge: q_2)\ t = [\![tr_2]\!]_{\mathsf{D}}\ q_2\ [\![tr_1]\!]_{\mathsf{M}}\ q_1\ t$$

*where* $q_1' = fmap\ (\lambda s\ q \to [\![tr_2]\!]_{\mathsf{D}}\ q\ s)\ q_1$

For phrasing the soundness theorem for $\cdot \circ_{\mathsf{DM}} \cdot$ we have to perform the same manipulation on the states of the MTT as in the definition of $\cdot \circ_{\mathsf{DM}} \cdot$ itself: we have to run the DTT on the accumulation parameters, thus transforming $q_1$ into $q_1'$ akin to the transformation in $nested$.[3]

Note that the three theorems stating the soundness of the composition of tree transducers can be read as fusion rules. Read from right to left, these rules describe how to fuse two transducer runs into a single one. Using GHC's rewrite rules facility [17], these fusion rules can be implemented as optimisation rules in GHC.

[3] Technically, the initial state of a tree transducer is part of the transducer itself. Thus, the construction of the initial state $(q_1' :\wedge: q_2)$ for the compound transducer is actually part of the composition construction.

## 7. Macro Tree Transducers with Regular Look-Ahead

MTTs are quite expressive, more expressive than one might consider after a first glance. As illustrated in Section 3 and 4, MTT emerge as a generalisation of DTT. As such, MTTs also retain the top-down flow of state information from DTTs. Naturally, there is also bottom-up version of DTTs, unsurprisingly called *bottom-up tree transducers* (UTTs). The expressive power of DTTs and UTTs is incomparable [7], i.e. there are transformations that can be expressed using one of the two kinds of transducer but not the other. Since MTTs generalise DTTs they can express any transformation expressible as a DTT. Surprisingly, however, MTTs can also express all transformations expressible as a UTTs[4] [9].

The underlying reason is that MTTs are closed under addition of *regular look-ahead* [9]: roughly speaking, an MTT with regular look-ahead (MTTL) is an MTT which has access to additional information that is provided by a second automaton that propagates states bottom-up. Intuitively, the run of such an MTTL can be thought of as a two-phase computation: in the first phase, the bottom-up tree automaton is run, annotating each node of the tree with the state that it computes for it. Afterwards, in the second phase, the MTT is run on this annotated tree and has therefore access to the state information that was provided by the bottom-up automaton in the first phase.

### 7.1 An Example

To illustrate why such a look-ahead is useful, we reconsider the inlining transformation from Section 3. We want to make this transformation a bit smarter by only performing the inlining for bound variables occurring exactly once in their scope. That means, when transforming a let binding, we first compute the number of occurrences of the bound variable in the scope of the let binding. This counting of variable occurrences is a bottom-up computation.

*Bottom-up tree automata* (UTA), also know as bottom-up tree acceptors, only propagate state information (upwards) without performing a transformation. In Haskell such an automaton is represented by the following type:

$$\mathbf{type}\ State_{\mathsf{U}}\ f\ q = f\ q \to q$$

The alert reader will notice that this is exactly the type for $f$-algebras with carrier type $q$. And indeed running a UTA on a term is simply the fold of this algebra:

$$\begin{aligned}&[\![\cdot]\!]_{\mathsf{U}}^{\mathsf{S}} :: Functor\ f \Rightarrow State_{\mathsf{U}}\ f\ q \to \mu f \to q\\ &[\![st]\!]_{\mathsf{U}}^{\mathsf{S}} = fold_{\mu}\ st\end{aligned}$$

The UTA that counts free occurrences of variables is then defined as follows:

$$\begin{aligned}&St_{\mathsf{numFV}} :: State_{\mathsf{U}}\ Sig\ (Map\ Var\ Int)\\ &St_{\mathsf{numFV}}\ (Add\ x\ y) &&= x \oplus y\\ &St_{\mathsf{numFV}}\ (Val\ \_) &&= \emptyset\\ &St_{\mathsf{numFV}}\ (Let\ v\ x\ y) &&= x \oplus (y \setminus v)\\ &St_{\mathsf{numFV}}\ (Var\ v) &&= \{v \mapsto 1\}\end{aligned}$$

where $\oplus$ adds up the variable counts of two mappings:

$$\begin{aligned}&\cdot \oplus \cdot :: Ord\ a \Rightarrow Map\ a\ Int \to Map\ a\ Int \to Map\ a\ Int\\ &x \oplus y = Map.unionWith\ (+)\ x\ y\end{aligned}$$

[4] Both for the case of unrestricted transducers, and for the case of total, deterministic transducers, which we are interested in here.

But we do not want to actually run this automaton on its own; we want to combine it with a variant of an MTT that can read the state information provided by a UTA. Before we do this, let us step back and have a look at the type representing MTTs again:

$$\textbf{type}\ \mathit{Trans}_{\mathsf{M}}\ f\ q\ g = \forall\ a\ .\ q\ a \rightarrow$$
$$f\ (q\ (g^*\ a) \rightarrow a) \rightarrow g^*\ a$$

The type of MTTs with regular look-ahead (MTTLs) has an additional type variable $p$ that refers to the state space of the UTA that provides the bottom-up state information:

$$\textbf{type}\ \mathit{Trans}^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g = \forall\ a\ .\ q\ a \rightarrow$$
$$f\ (q\ (g^*\ a) \rightarrow a, p) \rightarrow g^*\ a$$

The type $p$ is added in only one place, viz. paired with the type of input variables. This gives the transduction rules the ability to observe the state of the successor nodes of the current node. Before we consider the semantics, let us look at an example. We shall now implement the smart inline translation that only inlines let bindings whose variable occurs exactly once in its scope:

```
$trans_{\mathsf{smart}} :: \mathit{Trans'}^{\mathsf{L}}_{\mathsf{M}}\ \mathit{Sig}\ (\mathit{Map}\ \mathit{Var})\ (\mathit{Map}\ \mathit{Var}\ \mathit{Int})\ \mathit{Sig}$
$trans_{\mathsf{smart}}\ m\ (\mathit{Var}\ v) = \textbf{case}\ \mathit{Map.lookup}\ v\ m\ \textbf{of}$
                              $\mathit{Nothing} \rightarrow i\mathit{Var}\ v$
                              $\mathit{Just}\ e \quad \rightarrow e$
$trans_{\mathsf{smart}}\ m\ (\mathit{Let}\ v\ (x, \_)\ (y, \mathit{yvars})) =$
                   $\textbf{case}\ \mathit{Map.findWithDefault}\ 0\ v\ \mathit{yvars}\ \textbf{of}$
                     $0 \rightarrow y\ m$
                     $1 \rightarrow y\ (m\ [v \mapsto x\ m])$
                     $\_ \rightarrow i\mathit{Let}\ v\ (x\ m)\ (y\ m)$
$trans_{\mathsf{smart}}\ m\ (\mathit{Val}\ n) \qquad\qquad = i\mathit{Val}\ n$
$trans_{\mathsf{smart}}\ m\ (\mathit{Add}\ (x, \_)\ (y, \_)) = x\ m\ \text{`}i\mathit{Add}\text{`}\ y\ m$
```

The transducer coincides with the simpler original $trans_{\mathsf{inline}}$ in all cases except for $\mathit{Let}$. In the $\mathit{Let}$ case the transducer consults the bottom-up state $\mathit{yvars}$ of the $y$ argument, which contains the numbers of occurrences of free variables in the term represented by $y$. Depending on how many times the variable $v$ occurs, the transducer simply removes the let binding, performs inlining or performs no transformation at all.

Note that in the above definition, we did not use the type $\mathit{Trans}^{\mathsf{L}}_{\mathsf{M}}$ but instead $\mathit{Trans'}^{\mathsf{L}}_{\mathsf{M}}$, which lets us avoid explicit applications of $\mathit{Re}$ by packaging "naked" occurrences of $a$ into $g^*\ a$:

$$\textbf{type}\ \mathit{Trans'}^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g = \forall\ a\ .\ q\ (g^*\ a) \rightarrow$$
$$f\ (q\ (g^*\ a) \rightarrow g^*\ a, p) \rightarrow g^*\ a$$

As for previous transducer types, also this convenience type can be lifted to its original form by applying $\mathit{Re}$ in the right places:

```
$\Uparrow^{\mathsf{L}}_{\mathsf{M}} :: (\mathit{Functor}\ q, \mathit{Functor}\ f) \Rightarrow$
      $\mathit{Trans'}^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g \rightarrow \mathit{Trans}^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g$
$\Uparrow^{\mathsf{L}}_{\mathsf{M}}\ tr\ q\ t = tr\ (\mathit{fmap}\ \mathit{Re}\ q)$
                    $(\mathit{fmap}\ (\lambda(f, p) \rightarrow (\mathit{Re} \circ f, p))\ t)$
```

### 7.2 The Semantics

The easiest way to verbally describe the semantics of MTTLs is via a two-phase execution as alluded to above. At first the UTA is used to annotate the input term with the bottom-up state information, and then the MTTL is run on the annotated tree. However, the implementation in Haskell is best described as a fold:

```
$\mathit{alg}^{\mathsf{L}}_{\mathsf{M}} :: (\mathit{Functor}\ f, \mathit{Functor}\ q, \mathit{Functor}\ g) \Rightarrow$
  $\mathit{State}_{\mathsf{U}}\ f\ p \rightarrow \mathit{Trans}^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g \rightarrow \mathit{Alg}\ f\ (q\ \mu g \rightarrow \mu g, p)$
$\mathit{alg}^{\mathsf{L}}_{\mathsf{M}}\ st\ tr\ t = (\lambda q \rightarrow \mathit{join}\ (tr\ q\ t'), st\ (\mathit{fmap}\ \mathit{snd}\ t'))$
  $\textbf{where}\ t' = \mathit{fmap}\ (\lambda(\mathit{res}, p) \rightarrow (\mathit{res} \circ \mathit{fmap}\ \mathit{join}, p))\ t$
```

```
$[\![\cdot,\cdot]\!]^{\mathsf{L}}_{\mathsf{M}} :: (\mathit{Functor}\ g, \mathit{Functor}\ f, \mathit{Functor}\ q) \Rightarrow$
  $\mathit{State}_{\mathsf{U}}\ f\ p \rightarrow \mathit{Trans}^{\mathsf{L}}_{\mathsf{M}}\ f\ q\ p\ g \rightarrow q\ \mu g \rightarrow \mu f \rightarrow \mu g$
$[\![st, tr]\!]^{\mathsf{L}}_{\mathsf{M}}\ q\ t = \mathit{fst}\ (\mathit{fold}_{\mu}\ (\mathit{alg}^{\mathsf{L}}_{\mathsf{M}}\ st\ tr)\ t)\ q$
```

The algebra is defined so as to run the transducer allong with the bottom up state transition. To this end, the carrier of the algebra is a product, whose second component is the state space of the UTA. Note that in order to apply the transition function of the UTA, we have to project to this second component using $\mathit{fmap}\ \ \mathit{snd}$. The semantics of the MTTL is then given by the fold over this algebra followed by a projection to the first argument of the resulting pair.

We thus obtain the smart inlining transformation:

$$\mathit{inlineSmart} :: \mu \mathit{Sig} \rightarrow \mu \mathit{Sig}$$
$$\mathit{inlineSmart} = [\![\mathit{St}_{\mathsf{numFV}}, \Uparrow^{\mathsf{L}}_{\mathsf{M}}\ \ trans_{\mathsf{smart}}]\!]^{\mathsf{L}}_{\mathsf{M}}\ \emptyset$$

As we have mentioned in the beginning of this section, MTTs with regular look-ahead can be transformed to ordinary MTTs [9]. This astonishing property of MTTs is, however, mostly of theoretical interest. The MTT that one gets from this construction basically tries out every possible value of the state that the UTA would provide and then checks during the traversal which of the alternatives was actually the correct one. This happens at every node of the input tree. Thus, there is an exponential increase in the run-time in terms of the size of the state space of the UTA. Moreover, this construction does not work for infinite state spaces such as the state space $\mathit{Map}\ \ \mathit{Var}\ \ \mathit{Int}$, which we used in our example of the smart inlining transformation.

As a consequence, compositionality properties of MTTs do not immediately carry over to MTTLs represented in Haskell. Nevertheless, we can indeed generalise the composition $\cdot \circ_{\mathsf{DM}} \cdot$ to work on MTTs as well. For the converse order, a DTT followed by an MTTL, it is not clear whether such a composition can be defined in a sound way. The problem is that the UTA that is part of the MTTL has to be transformed properly such that it works for the source signature of the DTT. The situation, however, changes when we consider bottom-up tree transducers (UTTs) instead of DTTs. We can indeed effectively construct the composition of MTTLs and UTTs, and vice versa.

## 8. Discussion

### 8.1 Related Work

The transformations performed by tree transducers are closely related to attribute grammars [26]. In particular, the combination of a top-down and a bottom-up flow of information found in MTTs with regular look-ahead are likewise found in attribute grammars in the form of inherited and synthesised attributes. Viera et al. [33] have shown that attribute grammars can be embedded in Haskell as a library. Attribute grammars can be translated into recursive program schemes [6], which can be expressed as MTTs. Thus, MTTs subsume attribute grammars. Moreover, *attributed tree transducers* [10], a class of tree automata that has been introduced to study the properties of attribute grammars has been shown to be subsumed in expressivity by MTTs [11].

Upwards and downwards accumulations [12, 13] generalise the *scanl* and *scanr* functions on lists to regular data types. The propagation of state information in a tree automaton follows the same mechanics. As recently pointed out by Gibbons [14], there is also a strong connection between accumulations and attribute grammars: complete attribute evaluation can be expressed as an upwards accumulation followed by a downwards accumulation. The same idea is used in the definition of MTTs with regular look-ahead.

Our work on representing tree transducers in Haskell was influenced by the category theoretic definition of bottom-up tree transducers of Hasuo et al. [15]. For their definition, the authors introduce the idea of representing the placeholder variables in transduction rules as naturality conditions. We use the same idea in our representation, merely replacing naturality with parametric polymorphism. Jürgensen [18, 19] gives a categorical definition of top-down transducers and proves the soundness of their composition.

### 8.2 Future Work

When working with transducers in Haskell, one soon notices the need for an extension to monadic computations. For example, to make full use of the access path annotation transformation $trans_{\mathsf{path}}$, which we discussed in Section 5, one also needs a corresponding query function that, given an access path, returns the subtree at that access path. Naturally, such a transformation is not total as a given access path may not be present in an input tree. Thus, the (top-down) transduction rules should be instead of type

$$q \to f\ (q \to a) \to m\ (g^{*}\ a)$$

for some monad $m$. The semantics of DTTs can be easily generalised to this monadic form (using the $Traversable$ type class mentioned in Section 5). However, for the case of the $Maybe$ monad, this type corresponds to partial, deterministic DTTs, which are known to not be closed under composition [30]. But we conjecture that a slight change in the type recovers compositionality:

$$q \to f\ (q \to m\ a) \to m\ (g^{*}\ a)$$

This type allows us to observe the effect in a placeholder variable without using its result.

## References

[1] P. Bahr. Modular tree automata. In J. Gibbons and P. Nogueira, editors, *MPC '12*, volume 7342 of *LNCS*, pages 263–299. Springer, 2012.

[2] P. Bahr and T. Hvitved. Compositional data types. In *WGP '11*, pages 83–94. ACM, 2011.

[3] P. Bahr and T. Hvitved. Parametric compositional data types. In J. Chapman and P. B. Levy, editors, *MSFP '12*, volume 76 of *EPTCS*, pages 3–24, 2012.

[4] P. Bahr and T. Hvitved. `compdata` Haskell library, 2013. URL `http://hackage.haskell.org/package/compdata`. module `Data.Comp.MacroAutomata`.

[5] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007. Draft.

[6] B. Courcelle and P. Franchi-Zannettacci. Attribute grammars and recursive program schemes I. *Theor. Comput. Sci.*, 17(2):163 – 191, 1982.

[7] J. Engelfriet. Bottom-up and top-down tree transformations— a comparison. *Math. Syst. Theory*, 9(2):198–231, 1975.

[8] J. Engelfriet and S. Maneth. The equivalence problem for deterministic MSO tree transducers is decidable. *Inf. Process. Lett.*, 100(5):206 – 212, 2006.

[9] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. System Sci.*, 31(1):71 – 146, 1985.

[10] Z. Fülöp. On attributed tree transducers. *Acta Cybernet.*, 5:261–279, 1981.

[11] Z. Fülöp and H. Vogler. A characterization of attributed tree transformations by a subclass of macro tree transducers. *Theory Comput. Syst.*, 32(6):649–676, 1999.

[12] J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *MPC '93*, volume 669 of *LNCS*, pages 122–138. Springer, 1993.

[13] J. Gibbons. Generic downwards accumulations. *Sci. Comput. Prog.*, 37(1-3):37–65, 2000.

[14] J. Gibbons. Accumulating attributes (for Doaitse Swierstra, on his retirement). In J. Hage and A. Dijkstra, editors, *Een Lawine van Ontwortelde Bomen: Liber Amicorum voor Doaitse Swierstra*, pages 87–102. 2013.

[15] I. Hasuo, B. Jacobs, and T. Uustalu. Categorical views on computations on trees (extended abstract). In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *ICALP '07*, volume 4596 of *LNCS*, pages 619–630. Springer, 2007.

[16] H. Hosoya. *Foundations of XML Processing – The Tree-Automata Approach*. Cambridge University Press, 2010.

[17] S. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop '01*, page 203, 2001.

[18] C. Jürgensen. *Categorical semantics and composition of tree transducers*. PhD thesis, Technischen Universität Dresden, 2003.

[19] C. Jürgensen and H. Vogler. Syntactic composition of top-down tree transducers is short cut fusion. *Math. Structures Comput. Sci.*, 14(2): 215–282, 2004.

[20] K. Knight and J. Graehl. An overview of probabilistic tree transducers for natural language processing. In A. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 3406 of *LNCS*, pages 1–24. Springer, 2005.

[21] A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and S. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *LNCS*, pages 146–157. Springer Berlin / Heidelberg, 1998.

[22] A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In A. Middeldorp and T. Sato, editors, *FLOPS '99*, volume 1722 of *LNCS*, pages 114–130. Springer, 1999.

[23] A. Kühnemann and J. Voigtländer. Tree transducer composition as deforestation method for functional programs. Technical report, Dresden University of Technology, 2001.

[24] K. Matsuda, K. Inaba, and K. Nakano. Polynomial-time inverse computation for accumulative functions with multiple data traversals. In *PEPM '12*, pages 5–14. ACM, 2012.

[25] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *FPCA '91*, volume 523 of *LNCS*, pages 124–144. Springer, 1991.

[26] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[27] W. C. Rounds. Mappings and grammars on trees. *Math. Syst. Theory*, 4(3):257–287, 1970.

[28] H. Seidl. Equivalence of finite-valued tree transducers is decidable. *Math. Syst. Theory*, 27(4):285–346, 1994.

[29] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[30] J. W. Thatcher. Transformations and translations from the point of view of generalized finite automata theory. In *STOC '69*, pages 129–142. ACM, 1969.

[31] W. Thomas. Handbook of formal languages, vol. 3. chapter Languages, automata, and logic, pages 389–455. Springer-Verlag, 1997.

[32] S. Tison. Tree automata and term rewrite systems. In L. Bachmair, editor, *RTA '00*, volume 1833 of *LNCS*, pages 27–30. Springer, 2000.

[33] M. Viera, S. D. Swierstra, and W. Swierstra. Attribute grammars fly first-class. In *ICFP '09*, pages 245–246. ACM Press, 2009.

[34] J. Voigtländer. Conditions for efficiency improvement by tree transducer composition. In S. Tison, editor, *RTA '02*, volume 2378 of *LNCS*, pages 57–100. Springer, 2002.

[35] J. Voigtländer. Formal efficiency analysis for tree transducer composition. *Theory Comput. Syst.*, 41(4):619–689, 2007.

[36] P. Wadler. Theorems for free! In *FPCA '89*, pages 347–359. ACM, 1989.

[37] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[38] Z. Ésik. Decidability results concerning tree transducers I. *Acta Cybernet.*, 5:1–20, 1981.

$$
\begin{array}{l}
\cdot \circ_{\mathsf{DL}} \cdot :: \forall\ f\ g\ h\ q_1\ q_2\ p\ .\ (\mathit{Functor}\ f, \mathit{Functor}\ g, \mathit{Functor}\ h, \mathit{Functor}\ q_1) \Rightarrow \\
\quad \mathit{Trans}_{\mathsf{D}}\ g\ q_2\ h \to \mathit{Trans}^{\mathsf{L}}_{\mathsf{M}}\ f\ q_1\ p\ g \to \mathit{Trans}^{\mathsf{L}}_{\mathsf{M}}\ f\ (q_1 :\wedge: q_2)\ p\ h \\
tr_2 \circ_{\mathsf{DL}} tr_1\ (q_1 :\wedge: q_2)\ t = (\!|tr_2|\!)_{\mathsf{D}}\ q_2\ (tr_1\ (\mathit{fmap}\ (\lambda a\ q_2' \to a\ q_2')\ q_1)\ (\mathit{fmap}\ \mathit{reshape}\ t)) \\
\quad \mathbf{where}\ \mathit{reshape} :: ((q_1 :\wedge: q_2)\ (h^*\ a) \to a, p) \to (q_1\ (g^*\ (q_2 \to a)) \to q_2 \to a, p) \\
\qquad\qquad\ \ \mathit{reshape}\ (f, p) = (\lambda q_1'\ q_2' \to f\ (\mathit{fmap}\ (\lambda s\ q''_2 \to (\!|tr_2|\!)_{\mathsf{D}}\ q''_2\ s)\ q_1' :\wedge: q_2'), p)
\end{array}
$$

**Figure 6.** Composition of an MTTL followed by a DTT.

## A. Proof of Lemma 1

**Lemma 1.** *Let* $e = \lambda z\ q \to Re\ (z\ q)$ *and* $b = alg_{\mathsf{D}}\ tr$ *for some* $tr$. *Then the following holds for all* $x$ *and* $q$*:*

$$fold_\mu\ b\ (join\ x)\ q = join\ (fold_*\ e\ b\ (fmap\ (fold_\mu\ b)\ x)\ q)$$

*Proof of Lemma 1.* We proceed by induction on $x :: f^*\ \mu f$.

- Case $x = Re\ y$ for some $z :: \mu f$.

$$
\begin{array}{ll}
& join\ (fold_*\ e\ b\ (fmap\ f\ (fold_\mu\ b)\ (Re\ y))\ q) \\
= & \{\ \text{Definition of } fmap\ \} \\
& join\ (fold_*\ e\ b\ (Re\ (fold_\mu\ b\ y))\ q) \\
= & \{\ \text{Definition of } fold_*\ \} \\
& join\ (e\ (fold_\mu\ b\ y)\ q) \\
= & \{\ \text{Definition of } e\ \} \\
& join\ (Re\ (fold_\mu\ b\ y\ q)) \\
= & \{\ \text{Definition of } join\ \} \\
& fold_\mu\ b\ y\ q \\
= & \{\ \text{Definition of } join\ \} \\
& fold_\mu\ b\ (join\ (Re\ y))\ q
\end{array}
$$

- Case $x = In\ y$ for some $y :: f\ \mu f$.

$$
\begin{array}{ll}
& join\ (fold_*\ e\ b\ (fmap\ (fold_\mu\ b)\ (In\ y))\ q) \\
= & \{\ \text{Definition of } fmap\ \} \\
& join\ (fold_*\ e\ b\ (In\ (fmap\ (fmap\ (fold_\mu\ b))\ y))\ q) \\
= & \{\ \text{Definition of } fold_*\text{; functor law}\ \} \\
& join\ (b\ (fmap\ (fold_*\ e\ b \circ fmap\ (fold_\mu\ b))\ y)\ q) \\
= & \{\ \text{Definition of } b \text{ and } alg_{\mathsf{D}}\ \} \\
& join\ (join\ (tr\ q\ (fmap\ (fold_*\ e\ b \circ fmap\ (fold_\mu\ b))\ y))) \\
= & \{\ \text{Monad law: } join \circ join = join \circ fmap\ join\ \} \\
& join\ (fmap\ join\ (tr\ q\ (fmap\ (fold_*\ e\ b \circ fmap\ (fold_\mu\ b))\ y))) \\
= & \{\ \text{Parametricity; functor law}\ \} \\
& join\ (tr\ q\ (fmap\ ((join \circ) \circ fold_*\ e\ b \circ fmap\ (fold_\mu\ b))\ y)) \\
= & \{\ \text{Induction hypothesis}\ \} \\
& join\ (tr\ q\ (fmap\ (fold_\mu\ b \circ join)\ y)) \\
= & \{\ \text{Definition of } b \text{ and } alg_{\mathsf{D}}\ \} \\
& b\ (fmap\ (fold_\mu\ b \circ join)\ y)\ q \\
= & \{\ \text{Functor law; definition of } fold_\mu\ \} \\
& fold_\mu\ b\ (In\ (fmap\ join\ y))\ q \\
= & \{\ \text{Definition of } join\ \} \\
& fold_\mu\ b\ (join\ (In\ y))\ q
\end{array}
$$

□

## B. Composition of MTTLs

The definition of $\cdot \circ_{\mathsf{DL}} \cdot$ given in Figure 6 constructs the composition of an MTTL followed by a DTT.